Progress under bounded fairness

Hesselink, Wim H.

Published in:
Distributed computing

Publication date:
1999

Distrib. Comput. (1999) 12: 197–207

DISTRIBUTED COMPUTING

Progress under bounded fairness

Wim H. Hesselink

Department of Mathematics and Computing Science, University of Groningen, PO Box 800, NL-9700 AV Groningen, The Netherlands
(e-mail: wim@cs.rug.nl; http://www.cs.rug.nl/~wim)

Received: April 1998 / Accepted: March 1999

Summary. Progress is investigated for a shared-memory distributed system with a weak form of fault tolerance that allows processes to stop and restart functioning without notification. The concept of bounded fairness is introduced to formalize bounded delay under the assumption that each family of related processes continuously contains at least one active member. This is a generalization of wait-freedom, and also of a finitary form of weak fairness. Several useful proof rules are stated and proved. In a system with bounded fairness, a wait-free process can be constructed by forming a new process in which processes from the various families are scheduled in a round robin way. The theory is applied to prove progress within bounded delay for a linearizing concurrent data-object in shared memory. The safety properties of this algorithm have been treated elsewhere.

Key words: Bounded fairness – Concurrent data object – Fault tolerance – Memory management – Client server architecture

1 Introduction

The aim of this paper is to present a method for formally proving progress for a distributed system with a weak form of fault tolerance, together with a nontrivial application of this method.

The task of the system is distributed over a number of families of related processes. Every process is allowed to stop functioning without notification. Yet it is guaranteed that every invocation of the system is completed correctly within bounded delay, provided that every family always contains at least one active process. Processes do not perform erroneous actions. A stopped process may again become active, and if it does so, it restarts at the point where it stopped and with all its previous information. The fault tolerance refers to the fact that within a family of processes no more than one member needs to be active. This fault model is very weak: it does not allow "fault actions", e.g., cf. [1].

We consider progress assertions of the form "P leads to Q" where P and Q are predicates on the state. Such an assertion expresses that, if each of the families of processes continuously holds an active process and the system reaches a state where $P \wedge \neg Q$ holds, the system will subsequently reach a state where Q holds within bounded delay. So there must be a bound on the number of "rounds" needed to establish Q, which is independent of the run; here a "round" is a sequence of steps such that each family has at least one process that takes at least one step in it.

The progress property is formalized as bounded fairness with respect to a fairness set, which is a set of families of process names. Bounded fairness is stronger than unconditional fairness, cf. [3]. It is a form of unconditional (or weak) fairness in the sense that all processes are continuously enabled. It is stronger in the sense that progress is guaranteed within bounded delay if all processes are active enough, and that the assumption on the participation of the processes is weaker than usual (not all processes have to act but each family must have an acting process). Note that a stopped process is also enabled: it may restart again.

We regard the bounded delay property as the key issue. Bounded fairness generalizes wait-freedom, cf. [5], which requires that each process establishes its tasks in a bounded number of steps, independent of the actions of other processes. Bounded fairness allows the tasks of the algorithm to be distributed over different processes. In the case of round robin scheduling, the bounded delay property can be used to obtain bounds on the number of steps, thus enabling actual estimates of the time complexity.

Moreover, if one has a system with bounded fairness properties, one can construct processes with the corresponding wait-free properties by combining members from different families. Bounded fairness thus enables a separation of concerns that can be crucial for successful design.

We present methods to prove bounded fairness that are inspired by UNITY [2]. These methods were developed for (and are here applied to) the system presented in [8]. This is a linearizing concurrent data-object in shared memory. It consists of a number of *client* processes that concurrently issue invocations to the data-object, and four families of server processes that linearize and treat these invocations, update the value of the data-object, and deliver the results of the invocations to the clients.

The design goes back to [4,6]. These papers present wait-free solutions, in which different tasks cannot be delegated to different processes. Also, they use only one region of shared memory: the invocations and the resulting new states are placed at the same address. This leads to the requirement that the data object must be deterministic (the new state must be a function of the old state and the invocation), and that the register where the state of the object is written must be *safe* (it is allowed that different processes write the same new value concurrently into it). So there are three reasons for the new design: separation of concerns, elimination of determinacy, and elimination of safe registers.

The progress requirement of the system is that every invocation of an active client terminates within bounded delay provided that each family of server processes contains at least one active member.

For simplicity of the example, we do not treat memory management here. So we prove bounded fairness for the system under assumption of bounded fairness for memory management. This shows that the formalism can also be used for specification purposes. Actually, the theorem that memory management also makes progress with bounded fairness is more challenging, but we have been forced to omit it because of the size of its proof and the large number of relevant but boring details.

Overview

This paper is organized as follows. In Sect. 2, we define bounded fairness and give proof rules to infer it. These rules are counterparts for bounded fairness of some of the UNITY rules for weak fairness, cf. [2]. In this Section we also show how to construct wait-freedom in a system with bounded fairness. The process model and the repertoire of elementary instructions are described in Sect. 3. In Sect. 4, we give the specification of our application, the concurrent data-object of [8].

In Sect. 5 we describe the principal part of our distributed system, which consists of three families of processes: *clients*, *linearizers* and *appliers*. The invocations of the clients are linearized by the linearizers and treated by the appliers. The proof that this principal part satisfies the safety requirements of the specification is sketched in [8]. It is based on a mechanical proof [10] of safety for the total design, including memory management, where more than a hundred invariants have been verified mechanically. Our aim in this paper is to prove progress under assumption of the safety properties proved before and the progress properties of memory management that are specified in 5.4.

In Sect. 6, we prove the progress assertion for the system: every invocation of an active client leads to a configuration where the invocation has been treated and the client can invoke again. We have to rely on invariants of the system that were proved in [8]. We draw conclusions in Sect. 7.

2 Bounded fairness

In this Section we develop the theory of bounded fairness. We first describe a general set-up of distributed systems with shared memory. Then we give the definition of bounded fairness and present a small example and some special cases. Seven proof rules for bounded fairness are then stated and proved, followed by some corollaries. We finally show how bounded fairness can be used to construct wait-freedom.

2.1 Distributed systems and bounded fairness

A distributed system with shared memory consists of a set of named sequential processes that communicate by means of shared variables. This is formalized in the following (standard) way. The *configuration* consists of the values of the shared variables together with the values of the private variables, including the instruction pointers of the processes. We speak of configuration (instead of global state) to distinguish it from the state of the object as used in Sect. 4.

We use interleaving semantics. An *execution* of length n is a sequence of $n+1$ pairs $\langle x.i, q.i\rangle$ with $0 \leq i \leq n$ such that $x.i$ is a configuration and $q.i$ is a process name for every index i, and that an action of process $q.i$ can make a transition from configuration $x.i$ to configuration $x.(i+1)$ whenever $i < n$. Two executions (of lengths m and n) can be composed iff the final pair of the first execution equals the starting pair of the second execution. The composition is the catenation of the two executions with one of the matching pairs deleted; so it has length $m+n$.

The system description contains a set of *initial* configurations. A configuration is called *reachable* iff it occurs in an execution that starts in an initial configuration.

For us a *predicate* is a boolean function of the configuration. A predicate is called an *invariant* iff it holds in all reachable configurations. It is called *stable* (or inductive) iff it is preserved under every action.

Clearly, every stable predicate that holds initially is an invariant, and every predicate implied by an invariant is also invariant. These two facts form the main method to prove invariance, for the set of reachable configurations is usually not very tractable.

We now introduce the concept of bounded fairness.

Definition. A *fairness set* is a set of sets of process names. An execution $(i : 0 \leq i \leq n : \langle x.i, q.i\rangle)$ is called a *round* for fairness set L iff, for every $U \in L$, there is an index $i < n$ with $q.i \in U$. An execution is called *k-fair for L* iff it is a composition of k rounds for L (if $k > 0$). Every execution is regarded as 0-fair for L.

Let P and Q be predicates. We say that *P leads to Q under L within k* iff every execution k-fair for L that starts in a reachable configuration where P holds contains a configuration where Q holds. We use the notation $L : P \,\mathrm{o}\!\!\rightarrow Q$ within k. If the clause "within k" is omitted, we mean that "within k" can be added for some unspecified natural number k.

Informally speaking, each member U of the fairness set L is a set of processes that are supposed, collectively, to act often enough. An assertion $L : P \,\mathrm{o}\!\!\rightarrow Q$ means that, for every reachable configuration x where P holds, every execution that contains enough rounds and starts in x contains a configuration where Q holds. Moreover, the lower bound on the number of rounds is independent of x.

One may wonder why an execution is not called k-fair if, for every $U \in L$, it simply contains k actions of processes in U? The reason is that the actions from U may need actions of other processes to have taken place in order to be productive. The introduction of rounds has the effect that the actions of the families of processes must be sufficiently mixed, without imposing overspecific constraints.

Note that we use the same terminology as is used in UNITY, cf. [2], but that our notion of *leadsto* is different, since it contains bounds and mentions process names and fairness sets. Another difference with UNITY is that our processes have names and may have private variables. When both concepts apply, our concept of "leadsto" implies the UNITY concept of "leadsto", but not conversely.

Example. Consider a system with a shared integer variable `t` and a shared boolean variable `b` and the three processes

Inc: **if** `b` **then** `t` := `t` + 1 **fi** ,
Rev: `b` := *false* ,
Dec: `t` := `t` − 1 .

In *Inc* (and henceforth), the **if** statement means *skip* if the guard is false.

We may regard this declaration as an assignment section of a UNITY program. Then it satisfies: `t` = 1 leads to `t` = 0, because `t` is decremented often enough, since, because of *Rev*, it cannot be incremented infinitely often.

In our setting, we assume that each of the three processes *Inc*, *Rev*, and *Dec* repeats the corresponding command infinitely often. Consider the fairness set $L = \{\{Rev\}, \{Dec\}\}$. This specifies that *Rev* and *Dec* each are executed often enough. Yet, `t` = 1 does not lead to `t` = 0 in bounded fairness. In fact, in the first round, process *Rev* makes `b` = *false*, but there is no upper bound for the number of applications of *Inc* preceding *Rev*. Therefore, there is no upper bound for the number of times *Dec* must be executed to get `t` = 0.

If we replace the guard of *Inc* by, say, `b` $\wedge$ `t` < 9, we do have that `t` = 1 leads to `t` = 0 with respect to L. If we then replace L by $L = \{\{Rev, Dec\}\}$, it is false again, since there is no guarantee that *Rev* is ever executed, or that *Dec* is executed often enough. □

The definition of bounded fairness has two special cases worth mentioning. In the first case, L is the set of the singleton sets $\{q\}$ where q ranges over all process names. Now, an execution is a round if and only if every process acts in it at least once. This is the case of bounded fairness with *fault intolerance*, the form of bounded fairness we proposed in [7].

In the second special case, $L = \{\{q\}\}$ for some fixed process q. Here a round is an execution in which process q acts at least once. So, $L : P \,\mathrm{o}\!\!\rightarrow Q$ means that process q establishes Q starting in a configuration where P holds in a bounded number of steps, regardless of the actions of the other processes. This is the case of wait-freedom, as proposed in [5].

2.2 Proof rules

We now present and prove a number of rules about bounded fairness that are needed (and sufficient) to prove bounded fairness in our application. In these rules we use fairness sets L and M, and predicates P, Q, R, P', Q'. Most of the rules state that some *leadsto* relation can be inferred from other *leadsto* relations. We have two starting rules.

If U is a set of process names, we write $U : P \rhd Q$ to denote that, for every process $q \in U$, every action of q that starts in a reachable configuration where P holds ends in a configuration where Q holds. If the set U is omitted, we mean relation $\rhd$ to hold for the set of all processes.

Rule 0 (implication). If predicate P is stronger than Q, then $L : P \,\mathrm{o}\!\!\rightarrow Q$ within 0. □

Rule 1 (step). Assume $P \wedge \neg Q \rhd P \vee Q$ and $U : P \wedge \neg Q \rhd Q$. Then we have $\{U\} : P \,\mathrm{o}\!\!\rightarrow Q$ within 1.

Proof. Let $(i : 0 \leq i \leq n : \langle x.i, q.i\rangle)$ be a round for fairness set $\{U\}$, which starts in a reachable configuration where P holds. It suffices to prove that the round contains a configuration where Q holds. Since it is a round for $\{U\}$, there exists $j < n$ with $q.j \in U$. If there is an index $i \leq j$ such that Q holds in $x.i$ we are done. Otherwise we use $P \wedge \neg Q \rhd P \vee Q$ and induction to prove that all configurations $x.i$ with $i \leq j$ satisfy $P \wedge \neg Q$. In particular, $P \wedge \neg Q$ holds in $x.j$. Then $U : P \wedge \neg Q \rhd Q$ implies that Q holds in $x.(j + 1)$. □

Rule 2 (monotony). Assume that $L \subseteq M$, that $L : P \,\mathrm{o}\!\!\rightarrow Q$ within k, that P' is stronger than P, that Q is stronger than Q', and that $k \leq m$. Then we have $M : P' \,\mathrm{o}\!\!\rightarrow Q'$ within m.

Proof. Every execution m–fair for M is also k–fair for L. So, if such an execution starts in a reachable configuration where P' holds, it starts in a reachable configuration where P holds, and hence contains a configuration where Q holds and where therefore Q' holds. □

Rule 3 (disjunction). Let $(i :: P.i)$ be a family of predicates such that $L : P.i \,\mathrm{o}\!\!\rightarrow Q$ within k for all i. Then we have $L : (\exists\, i :: P.i) \,\mathrm{o}\!\!\rightarrow Q$ within k.

Proof. Let h be a k-fair execution for L that starts in a configuration where $(\exists\, i :: P.i)$ holds. Then there exists i such that $P.i$ holds in this configuration. Since $L : P.i \,\mathrm{o}\!\!\rightarrow Q$, the execution contains a configuration where Q holds. □

Remark. It must be noted (as pointed out by a referee) that Lemma 3 becomes false when the two clauses "within k" are omitted and i ranges over an infinite set. On the other hand, if i ranges over a finite set, Lemma 3 implies its variation in which the clauses "within k" are omitted. The point is that each i may need a different k, but if the range of i is finite, the greatest k will do. □

Rule 4 (delegation). Let U be a (nonempty) finite set of processes such that $\{\{q\}\} \cup L : P \,\mathrm{o}\!\!\rightarrow Q$ for every $q \in U$. Then we have $\{U\} \cup L : P \,\mathrm{o}\!\!\rightarrow Q$.

Proof. Since U is finite, we can choose a natural number k such that $\{\{q\}\} \cup L : P \,\mathrm{o}\!\!\rightarrow Q$ within k for every $q \in U$. Let h be a ($\#U \times k$)-fair execution for $\{U\} \cup L$, which starts in a reachable configuration where P holds. Since h is a composition of $\#U \times k$ rounds for $\{U\} \cup L$, there is a process $q \in U$ such that h is a composition of at least k rounds for $\{\{q\}\} \cup L$ (a version of the pigeonhole principle).

Therefore, h is a k–fair execution for $\{\{q\}\} \cup L$. It follows that h contains a configuration where Q holds. □

Remark. It is in Rule 4 that complexity suffers for fault tolerance, in the sense that the upper bound for $\{U\} \cup L : P \,\mathrm{o}\!\rightarrow Q$ is the product of $\#U$ with the upper bound for $\{\{q\}\} \cup L : P \,\mathrm{o}\!\rightarrow Q$. Indeed, all processes q may have to work for the goal Q. □

Rule 5 (transitivity). Assume that $L : P \,\mathrm{o}\!\rightarrow Q$ within k and $L : Q \,\mathrm{o}\!\rightarrow R$ within m. Then $L : P \,\mathrm{o}\!\rightarrow R$ within $k + m$.

Proof. We first note that, for $k,\ m \geq 0$, an execution is $k + m$-fair if and only if it can be split as a composition of a k-fair execution with an m-fair execution.

Let h be an execution, $(k + m)$-fair for L, that starts in a reachable configuration where P holds. Execution h is the composition of executions $h0$ and $h1$ such that $h0$ is k-fair and $h1$ is m-fair, both for L. Since execution $h0$ starts in a configuration where P holds, it contains a configuration where Q holds. Therefore $h0$ has a suffix $h2$ that starts in a configuration where Q holds. The executions $h2$ and $h1$ have a composition $h3$, which is m-fair for L and starts in a reachable configuration where Q holds. Therefore $h3$ contains a configuration where R holds. Since $h3$ is a suffix of h, this proves that h contains a configuration where R holds. □

Rule 6. Assume that $L : P \,\mathrm{o}\!\rightarrow\! Q$ within k and that $R \wedge \neg Q \ \rhd\ R$. Then we have $L : P \wedge R \,\mathrm{o}\!\rightarrow Q \wedge R$ within k.

Proof. Let $(i :: \langle x.i, q.i\rangle)$ be an execution k-fair for L, which starts in a reachable configuration where $P \wedge R$ holds. The execution has a configuration where Q holds. Let j be the first index such that Q holds in $x.j$. Using $R \wedge \neg Q \ \rhd\ R$ and induction in i, we get that R holds in all configurations $x.i$ with $i \leq j$. In particular, $Q \wedge R$ holds in $x.j$. □

In the remainder of this paper, we only use these Rules to prove bounded fairness. First, three corollaries.

Corollary 0. Assume that $L : P \wedge \neg R \,\mathrm{o}\!\rightarrow Q \vee R$ within k and $L : Q \,\mathrm{o}\!\rightarrow R$ within m. Then $L : P \,\mathrm{o}\!\rightarrow R$ within $k + m$.

Proof. We first observe that $L : P \wedge R \,\mathrm{o}\!\rightarrow Q \vee R$ within k because of Rule 0 and Rule 2. Using Rule 3, this is combined with the assumption to yield $L : P \,\mathrm{o}\!\rightarrow Q \vee R$ within k. A similar argument yields $L : Q \vee R \,\mathrm{o}\!\rightarrow R$ within m. Therefore the assertion follows from Rule 5. □

Corollary 1. Let *vf* be a state function with values in the natural numbers such that, for all natural numbers m,

$L :\ P \wedge \mathit{vf} \leq m \ \mathrm{o}\!\rightarrow\ Q \vee (P \wedge \mathit{vf} < m)$.

Then we have $L : P \ \wedge\ \mathit{vf} \leq m \,\mathrm{o}\!\rightarrow Q$.

Proof. This follows from Corollary 0, by induction in m. The base case uses that $\mathit{vf} < 0$ is false. □

When translated to UNITY, the next corollary is the PSP rule of [2] page 65 (PSP stands for progress safety progress).

PSP-rule. Let $L : P \,\mathrm{o}\!\rightarrow Q$ within k and $R \wedge \neg S \ \rhd\ S \vee R$. Then we have $L : P \wedge R \,\mathrm{o}\!\rightarrow S \vee (Q \wedge R)$ within k.

Proof. Monotony implies $L : P \,\mathrm{o}\!\rightarrow Q'$ within k for $Q' = S \vee Q$. We put $R' = S \vee R$. Then it is easy to verify that $R' \wedge \neg Q' \ \rhd\ R'$. Therefore Rule 6 implies $P \wedge R' \,\mathrm{o}\!\rightarrow Q' \wedge R'$ within k. Then the assertion follows by monotony . □

Remark. Conversely, Rule 6 follows from the PSP-rule by taking $S := Q \wedge R$. □

2.3 The construction of wait-free processes

Assume that we have a system that satisfies $L : P \,\mathrm{o}\!\rightarrow Q$ for predicates P and Q and a finite fairness set L. One can then easily construct a new process that, from a configuration where P holds, in a wait-free manner establishes a configuration where Q holds. This goes as follows. Take a finite set M of processes that contains a member of each set $U \in L$. Let process S be a parallel composition of the members of M scheduled in a round robin fashion. Then process S, starting in P, establishes Q in a bounded number of steps, independently of the actions of other processes, i.e., process S leads from P to Q in a wait-free manner. This is proved as follows.

Assume that $L : P \,\mathrm{o}\!\rightarrow Q$ within k. Consider an execution that starts in a configuration where P holds and that contains $(\#M \times k)$ actions of process S. This execution is a composition of k parts in each of which process S performs $\#M$ actions. Since process S performs the actions of the members of M in a round robin fashion, and since $M \cap U$ is nonempty for every $U \in L$, each of these parts is a round for fairness set L. Therefore, the execution itself is k-fair for L and, hence, contains a configuration where Q holds. This proves that S establishes Q from P within $(\#M \times k)$ actions.

It follows that, in design, bounded fairness can be used as a preparation for wait-freedom. This is important since bounded fairness allows delegation of subtasks to different (families of) processes whereas wait-freedom always requires that all tasks can be done by the same process. Thus, design for bounded fairness allows a separation of concerns precluded by the requirement of wait-freedom.

Remark. Note that the word "establishes" in the informal description of wait-freedom is wrong in that it suggests an unintended causality. The configuration where Q holds need not be reached by a step of S. □

3 The modelling and the repertoire

We now describe the process model in more detail. We consider looping sequential processes with numbered atomic instructions and a private variable *pc* as instruction pointer. This instruction pointer is needed since most of the invariants and progress predicates will refer to it. Indeed, the application we are aiming at has a fine-grain interleaving of the processes that forces us to use such low-level instruments. Since we have program locations, we may as well make (disciplined) use of **goto** commands.

We distinguish between actual variables and ghost variables, and between shared variables and private variables. The shared variables serve as main memory and for the communication between processes. The private variables are used for private computations and as pointers in the shared data space. Ghost variables are used in the specification and the proof of the algorithm. They are computationally irrelevant. Alternatively, they are called auxiliary variables or

history variables, see (e.g.) [11, 12]. Ghost variables are not allowed in guards and in the righthand side of assignments to ordinary variables. In concrete programs we give the assignments to ghost variables between braces, but we do not do so in idealized ones.

We also have to discuss the repertoire of atomic instructions. Every atomic instruction refers to at most one shared variable, cf. [11], preferably at most once. We have three types of shared integer variables t that can occur more than once in an instruction: counters, consensus variables and compare & swap variables. Such a variable t is called a special variable. It has one of the special instructions

t := t ± 1 {counter} ;
if t = 0 **then** t := v **fi** {consensus} ;
if t = u **then** t := v **fi** {compare & swap} .

Here, u and v are private variables. These instructions may be combined with modifications of private variables and ghost variables. A special variable t can also be reset by t := 0, but cannot be modified in other ways. Of course, it can occur in expressions. A consensus variable t can also be boolean instead of integral. In that case, the guard t = 0 is replaced by ¬t and the assignment t := 0 is replaced by t := *false*.

4 A concurrent data object in shared memory

The theory of Sect. 2 is applied to the construction of a concurrent data object as introduced in [4]. A *concurrent data object* is defined as a data structure shared by concurrent processes. So there are a number of client processes that may concurrently inspect or modify the state of the object. Such actions of the clients are called *invocations*. The results of these invocations must be compatible with some linear history of the object, but on the other hand the clients must be served with bounded delay. The object resides in a shared data space. It is passive, but there are families of server processes to handle the invocations.

The object is specified as follows, cf. [8]. The *abstract* data object is a quadruple $\langle W, w_0, U, R\rangle$ where W is the state space of the object, $w_0 \in W$ is the initial state, U is the input space (the set of invocations), and $R \subseteq W \times U \times W$ is the transition relation. If the object is invoked in state w with invocation u, it may go into state y iff $\langle w, u, y\rangle \in R$. We assume that relation R is total, i.e., for every pair $\langle w, u\rangle$ there exists y with $\langle w, u, y\rangle \in R$. The new state y need not be unique (as was required in [6]).

The *concurrent* data object $\langle W, w_0, U, R\rangle$ consists of a procedure that, conceptually, acts on one shared program variable w of type W and that could be specified by

proc *apply* (**in** p : *Cli*, $u : U$; **out** $y : W$)
{**pre** w = w', **post** w = $y \wedge \langle w', u, y\rangle \in R$}

for every initial value $w' \in W$. Here *Cli* is the set of client processes. A client process p calls procedure *apply* in the form *apply*(p, u, y) for the treatment of invocation u to obtain the new state y. So, p and u are input parameters and y is a result parameter. All clients may call *apply* concurrently and repeatedly. The problem is that concurrent calls must be treated each with bounded delay and yet, logically, in some linear order.

Example. The data object W could be a data base. Then invocations u would comprise queries in the data base as well as commands to modify the data. Presumably, we would not want to output the whole contents y of the data base in response to every invocation but only a tiny projection of it, e.g., the result of the query or a message "done". It is clearly useful that different clients can access the data base concurrently, and that they need not wait unnecessarily. □

The aim is to construct a distributed implementation of *apply*. Since relation R is given (and R is total), we may assume that a sequential implementation of R is available in the form of

proc *locapply* (**in** $u : U, w : W$; **out** $y : W$)
{ **post** $\langle w, u, y\rangle \in R$ }

This procedure can be used by the processes, provided that every atomic instruction mentions at most one shared variable and that concurrent reading and writing of shared variables of types U and W is avoided.

For each client process p, we define the local history $\beta.p$ to be the list of consecutive pairs $\langle u, y\rangle$ of corresponding invocations and results of process p. Conceptually, each client executes the infinite loop

$*$ [u := *arbitrary* ; *apply* (*self*, u, y) ; $\beta := \langle u, y\rangle : \beta$]

where *self* is the name of the executing process and where $\langle u, y\rangle : \beta$ is the list obtained by prefixing list β with $\langle u, y\rangle$. Note that we treat $\beta.q$ as a private (ghost) variable β of process q.

The requirement that concurrent invocations be treated, logically, in some linear order is called *linearizability*. It is formalized as follows. We require that the history of the object can be represented by an ordered list σ of triples $\langle p, u, y\rangle \in \mathit{Cli} \times U \times W$. The occurrence of triple $\langle p, u, y\rangle$ means that client p has performed an invocation u with resulting state y. This is formalized as follows.

Let the projection $\sigma \mid p$ of σ be defined recursively as the list of pairs given by $\varepsilon \mid p = \varepsilon$ for the empty list ε, and

$(\langle q, u, y\rangle : \sigma)|p$ = **if** $p = q$ **then** $\langle u, y\rangle : (\sigma|p)$ **else** $\sigma|p$ **fi** .

We then require that history σ is related to the local histories by the invariant

(Lin0) $\beta.p = \sigma \mid p$ for every client process p, whenever p is not invoking.

Here, "p is not invoking" means that p is at the start of the body of its loop: it has to choose a new value for u.

To express that σ is a legal sequential history of the abstract object, we define list σ to be *acceptable* iff we have the invariant

(Lin1) *acc*.σ ,

where predicate *acc* is defined as follows. Let *laSta*.σ be the last state of history σ, defined by *laSta*.$\varepsilon = w_0$ and *laSta*.$(\langle p, u, y\rangle : \sigma) = y$. Then *acc* is given by

acc.ε = *true*
acc.$(\langle p, u, y\rangle : \sigma)$ = *acc*.$\sigma \wedge \langle$*laSta*.$\sigma, u, y\rangle \in R$

Thus, the data object is said to be linearizing iff one can construct a ghost variable σ with initially $\sigma = \varepsilon$, that for every execution satisfies the invariants (Lin0) and (Lin1).

We model the repeated calls of procedure *apply* by means of a number of looping sequential processes. For each process, we number the atomic instructions and use an explicit instruction pointer *pc*, which is a private variable.

So, the programs of the client processes have the form

20 u := *arbitrary* ;
21 instructions to put u in shared memory ;
... and to obtain a *result* ;
... $\beta := \langle u, \mathit{result}\rangle : \beta$;
... other instructions ; **goto** 20 .

Now requirement (Lin0) is more explicitly expressed in

(Lin0') $pc.q = 20 \;\Rightarrow\; \beta.q = \sigma \mid q$.

We turn to aspects of the implementation of the data object. For the sake of separation of concerns, we split its task into four parts: linearization of the invocations, application of the transition relation of the object, memory management for the invocations, and memory management for the state of the object. For the sake of fault tolerance we delegate each of these four tasks to a family of server processes. We use a family *Lin* of linearizers to linearize the invocations, a family *App* of appliers to update the data object and return the result, and two families, *Coll* of collectors and *Distr* of distributors, for memory management.

The progress assertion to be proved is that every client $q0$ with $pc.q0 = 21$ arrives within bounded delay back at $pc.q0 = 20$, provided the client itself is active and each family of server processes continuously contains an active process. This condition is formalized as

(0) $L : pc.q0 = 21 \;\mathrm{o}\!\!\rightarrow\; pc.q0 = 20$,
where $L = \{\{q0\}, \mathit{Lin}, \mathit{App}, \mathit{Coll}, \mathit{Distr}\}$.

5 A linearizing design

We come to the description of the system of [8], as specified in Sect. 4. For the ease of presentation and to simplify the proof of progress, we make some minor modifications in the design. Below we give the programs for the processes in *Cli*, *Lin*, *App* and the specifications of the processes in *Coll* and *Distr*. As announced above, we do not treat memory management. In [8], the programs for the memory management processes (*Coll* and *Distr*) are too nondeterministic to guarantee progress. Therefore, in [9], they are changed in a minor way and then their progress properties are proved.

In each declaration of shared variables, we indicate which processes are allowed to modify the variable by adding the families of allowed modifiers between braces.

5.1 The shared data and the clients

We use two regions of shared memory, one for invocation values $u : U$, and one for state values $w : W$. Pointers into these regions are called *addresses* and *locations*, respectively. In both cases we use value 0 as the *nil* address; nothing is stored there.

We thus introduce finite sets *Ad* and *Lo* which do not contain 0, and $\mathit{Ad0} = \{0\} \cup \mathit{Ad}$ and $\mathit{Lo0} = \{0\} \cup \mathit{Lo}$, and the shared arrays

inv : **array** *Ad* **of** U $\{\mathit{Cli}\}$;
sta : **array** *Lo* **of** W $\{\mathit{App}\}$;
post : **array** *Ad0* **of** *Lo0* $\{\mathit{App}, \mathit{Coll}\}$.

Array *inv* holds the invocations. As indicated in the declaration, it is modified only by client processes. Array *sta* holds the states and is only modified by appliers. Array *post* points from an invocation address to the location of the resulting state. We require that $\mathit{post}.i = 0$ holds until the invocation $\mathit{inv}.i$ has been treated. It is only for convenience in the invariants that we allow index 0 for *post* (with the invariant $\mathit{post}.0 = 0$).

Recall that *Cli* is the set of names of client processes. We write $\mathit{Cli0} = \{0\} \cup \mathit{Cli}$ and use shared arrays

iloc : **array** *Cli* **of** *Ad0* $\{\mathit{Cli}, \mathit{Coll}\}$;
own : **array** *Ad0* **of** *Cli0* $\{\mathit{Cli}, \mathit{Coll}\}$.

If it is nonzero, $\mathit{iloc}.p$ is the address of the current invocation of process p. If it is nonzero, $\mathit{own}.i$ is the client with invocation at address i. We shall treat *own* as a ghost variable.

We now come to the program of the clients (see below). When a client q has obtained an invocation value u, it waits for an invocation address $i = \mathit{iloc}.q \neq 0$. It writes its value u at $\mathit{inv}.i$ and then sets a flag $\mathit{tolin}.i$ to indicate that i contains an invocation ready to be included in the linearization. It then waits until the invocation has been treated, i.e., until $\mathit{sl} = \mathit{post}.i \neq 0$. It reads the resulting state $\mathit{sta}.\mathit{sl}$ and then resets its *iloc* field to indicate that it can use a new address. For the purpose of garbage collection, it also lowers a flag $\mathit{isil}.i$ at address i.

So we use shared boolean arrays

tolin : **array** *Ad0* **of** *Bool* $\{\mathit{Cli}, \mathit{Lin}\}$;
isil : **array** *Ad0* **of** *Bool* $\{\mathit{Cli}, \mathit{Coll}\}$.

Truth of $\mathit{tolin}.i$ means that $\mathit{inv}.i$ is a waiting invocation, and $\mathit{isil}.i$ indicates that address i has an owner.

In this way, we arrive at program Client for the client processes. Recall that *self* is the name of the executing process. Client has the private variables u for the current invocation, i and *sl* as copies of shared information, and the ghost variable β mentioned in the specification. Variables i and *sl* are used instead of $\mathit{iloc}.\mathit{self}$ and $\mathit{post}.i$ to avoid that a single instruction has to access more than one shared variable. The result of the invocation is obtained in the read action 25, where ghost variable β is updated.

Client
20 u := *arbitrary* ;
21 i := $\mathit{iloc}.\mathit{self}$; **if** $i = 0$ **then goto** 21 **fi** ;
22 $\mathit{inv}.i := u$;
23 $\mathit{tolin}.i$:= *true* ;
24 *sl* := $\mathit{post}.i$; **if** *sl* = 0 **then goto** 24 **fi** ;
25 { $\beta := \langle u, \mathit{sta}.\mathit{sl}\rangle : \beta$ } ;
26 $\mathit{iloc}.\mathit{self}$:= 0 { $\mathit{own}.i := 0$} ;
27 $\mathit{isil}.i$:= *false* ; **goto** 20 .

Readers concerned about safety should refer to [8]. The problem of this paper is progress. Program Client contains two points where progress is threatened: it uses busy waiting at 21 and 24. We come back to this in Sect. 6.

5.2 Linearization

We introduce a family *Lin* of server processes for the linearization of the invocations. The task of each linearizer is to enqueue all pending invocations of clients. We provide each linearizer with private variables z : *Ad0* and s : *Cli* and the linearizer has the task to linearize invocation address z of client s. The clients must be treated fairly. We therefore provide a function *nextCli* to choose a new client. This function traverses the set *Cli* of clients in the sense that, if q executes s := *nextCli*(q, s) repeatedly, all elements of *Cli* are met in some order. Function *nextCli* has first argument q, so that the order may differ for different linearizers. Indeed, if different linearizers are concurrently active, it is advantageous to let them use different orders to avoid congestion.

From the abstract point of view, we linearize the invocations by enqueueing them in a shared ghost variable

ilist : **queue of** *Ad* $\{Lin, App\}$.

So, the idealized linearizer would execute the infinite loop

$*$ [z := *iloc*.s ;
  if *tolin*.z **then**
    ilist := *ilist* $+\!\!+ \langle z \rangle$;
    tolin.z := *false*
  fi ;
  s := *nextCli*(*self*, s)] .

We need no test $z \neq 0$ here, since we keep the invariant $\neg$ *tolin*.0. The operator $+\!\!+$ stands for concatenation of lists.

We implement *ilist* by a list with links represented by *nx* and a tail represented by *invTail* (*invHead* in [8]), according to the shared variable declarations

nx : **array** *Ad0* **of** *Ad0* $\{Lin, Coll\}$;
invTail : *Ad* $\{Lin\}$.

The representation invariants for *ilist* are given in Sect. 6.3.

In view of the rules for occurrence of shared variables in atomic commands, we provide each linearizer with a private variable y as a copy of the shared variable *invTail*. The abstract assignment *ilist* := *ilist* $+\!\!+ \langle z \rangle$ is represented concretely by

y := *invTail* ;
{ *nx*.y = 0 ? } *nx*.y := z ;
{ *invTail* = y ? } *invTail* := z .

Since other linearizers may be active concurrently, this code is only applicable in so far as the assertions between braces hold (this is merely the intuition, we do not intend to give the question marks a formal meaning). The situation is sketched in diagram Fig. 1 where a solid arrow represents the initial value of a shared variable and a dashed arrow represents its new value.

In order to avoid that the collector processes recycle addresses prematurely, we introduce a shared array *cnt* for reference counting, declared by

cnt : **array** *Ad* **of** *int* $\{Lin, App\}$.

In this way we arrive at the following program where *ilist* is merely a ghost variable:

Linearizer

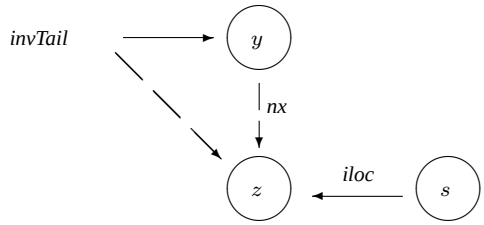

Fig. 1.

29 y := *invTail* ;
30 *cnt*.y := *cnt*.y + 1 ;
31 **if** $y \neq$ *invTail* **then goto** 38 **fi** ;
32 z := *iloc*.s ;
33 **if** $\neg$*tolin*.z **then**
  s := *nextCli*(*self*, s) ; **goto** 38 **fi** ;
34 **if** *nx*.y = 0 **then**
  nx.y := z { *ilist* := *ilist* $+\!\!+ \langle z \rangle$ }
  s := *nextCli*(*self*, s) **fi** ;
35 z := *nx*.y ;
36 *tolin*.z := *false* ;
37 **if** *invTail* = y **then** *invTail* := z **fi** ;
38 *cnt*.y := *cnt*.$y - 1$; **goto** 29 .

The test at 31 is needed for the case that a collector recycles address $y.q$ when *pc*.q = 30. The guards of 34 and 37 are needed since several linearizers may be active concurrently. The special forms of the atomic commands 34 and 37 show that the shared variable *nx*.y is a consensus variable and that *invTail* is a compare & swap variable.

Note that a linearizer may stop functioning after executing the **then** part of 34. Then progress requires that another linearizer executes 35, 36, and the **then** part of 37. Such operational arguments will not appear in the proof of progress in Sect. 6, but they were essential for the design of the system.

5.3 Application

We introduce a family *App* of appliers, which concurrently compute and store the results of procedure *locapply* for invocations in *ilist*. So the queue *ilist* produced by the linearizers is consumed by the appliers. We use a shared variable *staHead* to stand for the head of queue *ilist* and assume that *post*.*staHead* is the location of the current state of the object. Therefore *nx*.*staHead* is the address of the invocation that is to be treated next.

An applier q can be active when it has a location *staloc*.$q \neq 0$ to hold a new state. Recall from Sect. 4 that σ is the shared ghost variable that holds the history of the object. We thus have the shared variables

staHead : *Ad* $\{App\}$;
staloc : **array** *App* **of** *Lo0* $\{App, Distr\}$;
σ : **list of** $Cli \times U \times W$ $\{App\}$.

The appliers use private variables *sm*, *sl* for locations, y, z for addresses, *linv* for an invocation, and *new* for a state, all according to the situation sketched in diagram Fig. 2, as explained below and formalized in program Applier.

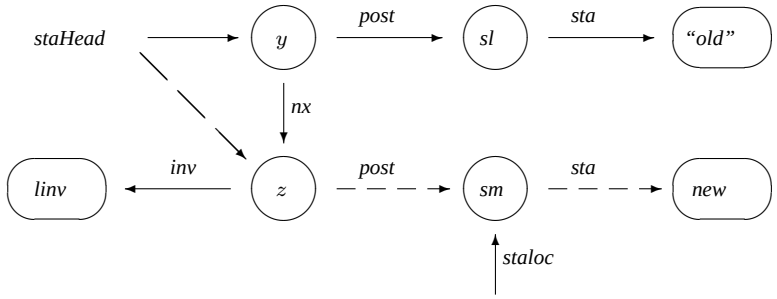

Fig. 2.

We first give an idealized code for the appliers, again only applicable when the assertions between braces hold.

$*$ [*sm* := *staloc.self* { *sm* $\neq 0$? } ;
y := *staHead* ; *z* := *nx.y* { *z* $\neq 0$? } ;
linv := *inv.z* ; *sl* := *post.y* ;
locapply(*linv*, *sta.sl*, *new*) ; *sta.sm* := *new* ;
{ *post.z* = 0 ? } *post.z* := *sm* ;
σ := $\langle$*own.z*, *linv*, *new*$\rangle$: σ ;
{ *y* = *staHead* ? } *staHead* := *z* ; *ilist* := *tail.ilist* ;
{ *post.z* = *sm* ? } *staloc.self* := 0] .

The first two question marks here are a matter of waiting. After that, the applier can perform a private computation of the next state *new*, which is then stored at *sta.sm*. The third question mark is more critical. Here the first applier "wins": assigns *sm* to *post.z* and extends σ accordingly (recall that *own.z* is the client that owns the invocation at z). At the fourth question mark, the first applier that comes there with current y moves *staHead* forward and removes the head from *ilist*. Finally, if location *sm* has been used, garbage collection is informed of the need of a new location.

The concrete program Applier is given below. Here, all potential interferences have been precluded. For this purpose we use some additional tests, and the shared variable

usob : **array** *Ad0* **of** *Bool* $\{App, Coll\}$.

Roughly speaking, *usob*.y indicates that address y is (or will be) an element of *ilist*. Array *usob* is used in the collectors, together with *isil* and *cnt*, to avoid premature garbage collection.

Applier
43 *sm* := *staloc.self* ; **if** *sm* = 0 **then goto** 43 **fi** ;
44 *y* := *staHead* ;
45 *cnt.y* := *cnt.y* + 1 ;
46 **if** *y* $\neq$ *staHead* **then goto** 57 **fi** ;
47 *z* := *nx.y* ; **if** *z* = 0 **then goto** 57 **fi** ;
48 *linv* := *inv.z* ;
49 *sl* := *post.y* ;
50 *locapply*(*linv*, *sta.sl*, *new*) ;
51 *sta.sm* := *new* ;
52 **if** *post.z* = 0 **then**
post.z := *sm* ;
{ σ := $\langle$*own.z*, *linv*, *new*$\rangle$: σ } **fi** ;
53 **if** *staHead* = *y* **then**
staHead := *z* { *ilist* := *tail.ilist* } **fi** ;
54 **if** *post.z* $\neq$ *sm* **then goto** 56 **fi** ;
55 *staloc.self* := 0 ;
56 *usob.y* := *false* ;
57 *cnt.y* := *cnt.y* $-$ 1 ; **goto** 43 .

With respect to progress, it should be noted that an applier may execute the **then** part of 52 and then stop functioning (for some time). In that case, another applier may have to execute the **then** part of 53, after an unproductive computation at 50 and skipping at 52 since it finds *post.z* $\neq$ 0.

It has been proved, cf. [8], that the system of the clients, linearizers and appliers, described here, preserves the invariants (Lin0) and (Lin1) of the specification in Sect. 4.

5.4 Specification of garbage collection

We finally specify the collectors (in *Coll*) and the distributors (in *Distr*). These processes have to supply the clients and the appliers with free addresses and locations, respectively, as formalized in the progress assertions

(1) $\{Coll\}$: *true* o$\rightarrow$ *iloc.q* $\neq 0$, for all $q \in Cli$,
(2) $\{Coll, Distr\}$: *true* o$\rightarrow$ *staloc.q* $\neq 0$,
for all $q \in App$.

On the other hand, collectors and distributors must preserve all invariants for *Cli*, *Lin*, *App*, described in [8]. In the mechanical proof of [10], we have verified this for the programs for *Coll* and *Distr* of [8].

It is easy to see that progress cannot be guaranteed if the sets *Ad* and *Lo* are too small in comparison with the sets of processes. Therefore, we assume that the sets *Ad* and *Lo* are large enough. In [9], we obtain lower bounds for the sizes of *Ad* and *Lo* for which the implementations of collectors and distributors provided satisfy the requirements (1) and (2). Thus, in the remainder of this paper, we can treat (1) and (2) as postulates.

6 Formal proof of progress

In this Section we prove progress assertion (0) of Sect. 4 under assumption of postulates (1) and (2). The global structure of the proof is as follows. Since (0) expresses progress for the *pc* of an active client, the main argument follows program Client. Client waits at two points: at *pc* = 21 and *pc* = 24. Progress at 21 is shown by means of postulate (1). Progress at 24, however, requires activity of both linearizers and appliers. These activities are specified by separate progress assertions that are dealt with in separate subsections.

6.1 The global proof

We now give the global proof of progress assertion (0) of Sect. 4. Recall that it expresses that an arbitrary client $q0$ gets its invocation treated, and that it reads

(0) $L:\ pc.q0 = 21\ \mathrm{o}\!\!\rightarrow\ pc.q0 = 20$, where
$L = \{\{q0\}, Lin, App, Coll, Distr\}$.

In the proof of (0), we use the proof rules of Sect. 2 and postulates (1) and (2) above. We postpone the proofs of some derived proof obligations. We need many invariants that have been established for the proof of safety. Such invariants are called *old invariants* and can be found in [8].

The proof of (0) follows the instructions of program Client. It is easy to see that

$$pc.q0 = 21 \wedge iloc.q0 = 0 \ \rhd\ pc.q0 = 21 .$$

Therefore, by Rule 6, postulate (1) implies

(3) $\{Coll\}:\ pc.q0 = 21\ \mathrm{o}\!\!\rightarrow$
$pc.q0 = 21 \wedge iloc.q0 \neq 0$.

Since *pc* and i are private variables, and $iloc.q0$ is a consensus register that is reset only by $q0$ itself in instruction 26 (cf. [8]), we have the $\rhd$ relations

$pc.q0 = 21 \ \wedge\ iloc.q0 \neq 0$
$\ \rhd\ pc.q0 \in \{21, 22\} \wedge iloc.q0 \neq 0$;
$\{q0\}:\ pc.q0 = 21 \wedge iloc.q0 \neq 0 \ \rhd\ pc.q0 = 22$.

By Rule 1 this implies

$\{\{q0\}\}:\ pc.q0 = 21 \ \wedge\ iloc.q0 \neq 0\ \mathrm{o}\!\!\rightarrow\ pc.q0 = 22$.

By similar arguments we obtain

$\{\{q0\}\}:\ pc.q0 = 22\ \mathrm{o}\!\!\rightarrow\ pc.q0 = 23$;
$\{\{q0\}\}:\ pc.q0 = 23\ \mathrm{o}\!\!\rightarrow\ pc.q0 = 24 \wedge tolin.(i.q0)$.

Therefore Rule 5 yields

(4) $\{\{q0\}\}:\ pc.q0 = 21 \ \wedge\ iloc.q0 \neq 0\ \mathrm{o}\!\!\rightarrow$
$pc.q0 = 24 \ \wedge\ tolin.(i.q0)$.

Here we are at the main critical point of Client: busy waiting at 24 must not lead to unbounded delay. This point is treated by precisely specifying the progress requirements for linearizers and appliers, (5) and (6) below, and subsequently proving that these requirements are met.

An old invariant (Dq3) says that $i.q0 \neq 0$ when $21 < pc.q0 \leq 26$. So, in the postcondition of (4) we may add $i.q0 \neq 0$. For the moment we replace $i.q0$ by an arbitrary address $k \neq 0$. In Sect. 6.2 we prove for $k \neq 0$ that

(5) $\{Lin\}:\ tolin.k \wedge post.k = 0\ \mathrm{o}\!\!\rightarrow\ k = nx.invTail$.

In Sect. 6.3 we use postulate (2) to prove for every address $k \neq 0$ that

(6) $\{App, Coll, Distr\}:\ k = nx.invTail\ \mathrm{o}\!\!\rightarrow\ post.k \neq 0$.

By Rule 5 (and also using Rules 0, 2, 3), the formulas (5) and (6) combine to yield

$$L:\ tolin.k\ \mathrm{o}\!\!\rightarrow\ post.k \neq 0 .$$

On the other hand, since *pc* and i are private variables, we have

$pc.q0 = 24 \wedge i.q0 = k \wedge post.k = 0$
$\ \rhd\ \ pc.q0 = 24 \wedge i.q0 = k$.

By Rule 6 (and 2), these two facts combine and yield

$L:\ pc.q0 = 24 \wedge i.q0 = k \wedge tolin.k$
$\mathrm{o}\!\!\rightarrow\ pc.q0 = 24 \wedge post.(i.q0) \neq 0$.

Since k ranges over the finite set *Ad*, Rule 3 (disjunction) now implies

(7) $L:\ pc.q0 = 24 \wedge tolin.(i.q0)$
$\mathrm{o}\!\!\rightarrow\ pc.q0 = 24 \wedge post.(i.q0) \neq 0$.

For the last stretch, we use old invariants that express that $post.k$ is reset to 0 (by collectors) only when $k \neq iloc.q$, and that $iloc.q = i.q$ when $21 < pc.q \leq 26$. This implies that

$pc.q0 = 24\ \wedge\ post.(i.q0) \neq 0$
$\ \rhd\ \ pc.q0 \in \{24, 25\}\ \wedge\ post.(i.q0) \neq 0$.

Then, again using Rules 1 and 5, we easily obtain

(8) $\{\{q0\}\}:\ pc.q0 = 24 \wedge post.(i.q0) \neq 0$
$\mathrm{o}\!\!\rightarrow\ pc.q0 = 20$.

Finally, formula (0) follows by Rule 5 (and 2) from (3), (4), (7), and (8). Thus, it remains to prove the formulas (5) and (6).

6.2 Progress for linearization

In this subsection we treat proof obligation (5), which expresses that an address to be linearized is linearized within bounded delay. This is the joint responsibility of the family *Lin*. Therefore, the proof of (5) needs inspection of program Linearizer. We prove formula (5) for a fixed address $k1 \neq 0$.

By old invariants the precondition $tolin.k1 \wedge post.k1 = 0$ implies that $k1 = iloc.q1$ for $q1 = own.k1 \in Cli$. By Rule 3 (disjunction), it therefore suffices to prove $\{Lin\}:\ P1\,\mathrm{o}\!\!\rightarrow Q1$, where for given $q1 \in Cli$ the predicates $P1$ and $Q1$ are given by

$P1:\quad tolin.k1 \wedge post.k1 = 0 \wedge k1 = iloc.q1$,
$Q1:\quad k1 = nx.invTail$.

Since we have to establish $Q1$, it is useful to know that 34 is the only command that can assign a nonzero value to an element of *nx*. On the other hand we have the old invariant

(Cq2) $31 < pc.q \leq 37 \wedge nx.(y.q) = 0 \ \Rightarrow$
$y.q = invTail$.

This invariant implies that a linearizer q establishes $Q1$ whenever it executes 34 with $nx.(y.q) = 0$ and $z.q = k1$. As for the precondition of our proof obligation, it follows from some other old invariants that $P1$ is not falsified while $Q1$ is false:

(v0) $P1 \wedge \neg Q1\ \rhd\ P1$.

We now have to prove progress towards a situation where a linearizer q executes 34 with $nx.(y.q) = 0$ and $z.q = k1$. Unfortunately, any given linearizer q may always find $nx.(y.q) \neq 0$ (individual starvation of a linearizer). It is only collectively that the task will be done.

For every linearizer q we introduce the private ghost variable $gs.q$ as the number of applications of *nextCli* (in

33 or 34) needed to reach $s.q = q1$. Put *pI = #Cli*. We let variable $gs.q$ be modified only in the **then** parts of 33 and 34, according to the additional (ghostly) instruction

$\{$ **if** $gs = 0$ **then** $gs := pI - 1$ **else** $gs := gs - 1$ **fi** $\}$.

It satisfies the invariants $0 \leq gs.q < pI$ and

$gs.q = 0 \equiv s.q = q1$.

The text of program Linearizer may suggest that $z.q = iloc.(s.q)$ when $32 < pc.q \leq 34$, but this is not necessarily the case ($iloc.(s.q)$ can be modified). We therefore adapt $gs.q$ by introducing a state function $vs.q$ given by

$vs.q =$ **if** $gs.q = 0 \ \wedge \ 32 < pc.q \leq 34 \ \wedge \ z.q \neq iloc.q1$
then pI **else** $gs.q$ **fi** .

Function $vs.q$ can only increase in the **then** parts of 33 and 34 when $gs.q = 0 \ \wedge \ z.q = iloc.q1$ holds, or when $iloc.q1$ is modified.

Aiming at an application of Corollary 1 of Sect. 2, we define the variant function

$$vf = (\sum q \in Lin :: vs.q) + \#(nx.invTail \neq 0) \ ,$$

where, for boolean b, we write $\#b$ to denote 1 if b holds and 0 otherwise.

Since *vf* is bounded, our proof obligation $\{Lin\} : P1 \ \mathrm{o}\!\!\rightarrow Q1$ follows by Corollary 1 from

$\{Lin\} : \ P1 \wedge vf \leq m \ \mathrm{o}\!\!\rightarrow$
$Q1 \vee (P1 \wedge vf < m)$.

It is not hard to prove the safety property

(v1) $P1 \wedge vf \leq m \ \rhd \ Q1 \vee vf \leq m$.

We have that *invTail* is modified only in 37, whereas old invariants imply that, if $pc.q = 37$ and $invTail = y.q$, then $nx.(y.q) \neq 0$ and $nx.(z.q) = 0$. Due to the second summand of *vf*, this implies that, for any constant address k,

(v2) $vf \leq m \wedge invTail = k \ \rhd$
$vf < m \vee invTail = k$.

For any fixed address k we introduce the predicates

$P2 : \quad P1 \wedge vf \leq m \wedge invTail = k$,
$Q2 : \quad Q1 \vee (P1 \ \wedge \ vf < m)$.

By disjunction and delegation (Rules 3 and 4), it now suffices to prove the progress assertion $\{\{q\}\} : P2 \, \mathrm{o}\!\!\rightarrow Q2$ for every $q \in Lin$ and $k \in Ad$.

The results (v0), (v1), (v2) combine to yield

(v3) $P2 \wedge \neg Q2 \ \rhd \ Q2 \vee P2$.

By inspection of program Linearizer and Rules 1, 2, 3, 5, we obtain, for any $q \in Lin$, $k \in Ad$,

$\{\{q\}\} : \ true \ \mathrm{o}\!\!\rightarrow \ invTail \neq k \vee (y.q = k \ \wedge \ pc.q = 31)$.

The PSP–rule with (v3) then implies

$\{\{q\}\} : \ P2 \ \mathrm{o}\!\!\rightarrow \ Q2 \vee (P2 \ \wedge \ y.q = k \ \wedge \ pc.q = 31)$.

Again by inspection of program Linearizer, one can prove that

$\{q\} : \ P2 \ \wedge \ y.q = k \ \wedge \ pc.q = 31$
$\rhd \ \ y.q = k \ \wedge \ pc.q = 32$;
$\{q\} : \ y.q = k \ \wedge \ pc.q = i \ \wedge \ 31 < i < 37$
$\rhd \ \ y.q = k \ \wedge \ pc.q = i + 1$;
$\{q\} : \ P2 \ \wedge \ y.q = k \ \wedge \ pc.q = 37 \ \rhd \ invTail \neq k$.

Now Rule 1 with (v3) and transitivity implies

$\{\{q\}\} : \ P2 \ \wedge \ y.q = k \ \wedge \ pc.q = 31 \ \mathrm{o}\!\!\rightarrow \ Q2$.

By Corollary 0, this implies $\{\{q\}\} : P2 \, \mathrm{o}\!\!\rightarrow Q2$ as required. This concludes the proof of (5).

6.3 Progress for application

The subsection is devoted to the proof of formula (6) that $nx.invTail = k$ for $k \neq 0$ leads to $post.k \neq 0$. Note that this expresses that every invocation address k, once enqueued, gets an associated object state within bounded delay. In order to get the postcondition of (6) we use the old invariant

(Bq2) $post.staHead \neq 0$,

In order to prove (6), it therefore suffices to prove for $k \neq 0$ that

$$\{App, Coll, Distr\} : \ k = nx.invTail \ \mathrm{o}\!\!\rightarrow \ staHead = k \ .$$

The addresses *staHead* and *nx.invTail* are connected via the ghost variable *ilist* and the representation invariants

(Lq0) $ilist_0 = staHead$;
(Lq1) $0 < i < \#ilist \ \ \Rightarrow \ \ nx.ilist_{i-1} = ilist_i \neq 0$;
(Lq2) $last.ilist =$ **if** $nx.invTail = 0$ **then** $invTail$
else $nx.invTail$ **fi** .

Here the elements of *ilist* are subscripted and numbered from 0, and *last.ilist* is its last element.

Since *ilist* has length bounded by *#Ad* (and does not contain address 0), it suffices to prove that *staHead* moves along the list, i.e., that for every $m \neq 0$,

(9) $\{App, Coll, Distr\} : \ nx.staHead = m \ \mathrm{o}\!\!\rightarrow \ R$,
where $R : \quad staHead = m$.

Strictly speaking, the reduction to proof obligation (9) requires an application of Corollary 1, with the index of address m in *ilist* as variant function *vf*.

By Rule 4 (delegation), proof obligation (9) reduces to the obligation to prove, for every $q \in App$,

(10) $\{\{q\}, Coll, Distr\} : \ nx.staHead = m \ \mathrm{o}\!\!\rightarrow \ R$.

Formula (10) depends on modification of *staHead*. Now *staHead* is modified only at 53 of program Applier. We have the difficulty that predicate $staHead = k$ is not stable. So we cannot guarantee that an applier q proceeds to $pc.q = 53$ with $y.q = staHead$. We are saved by the observation that modification of *staHead* also establishes postcondition R. This goes as follows. Old invariants imply that $z.r = nx.(y.r) \neq y.r$ whenever $pc.r = 53$ for any applier r. Using some more old invariants, we get, for arbitrary k and $m \neq 0$,

(w0) $staHead = k \ \wedge \ nx.k = m$
$\rhd \ (staHead = k \ \wedge \ nx.k = m) \ \vee \ R$.

We now start at the other end for an arbitrary applier q and address $m \neq 0$. By inspection of program Applier, it follows with Rules 1 and 5 that

$\{\{q\}\}$: *true* o→ *pc*.q = 43 .

Using postulate (2) and Rule 6, we get

$\{Coll, Distr\}$: *pc*.q = 43 o→ *pc*.q = 43 ∧ *staloc*.$q \neq 0$.

Since the consensus variable *staloc*.q is only reset by q itself in 55, one can use Rule 1 to prove

$\{\{q\}\}$: *pc*.q = 43 ∧ *staloc*.$q \neq 0$ o→ *pc*.q = 44 .

It is easy to see that

$\{\{q\}\}$: *pc*.q = 44 o→ *pc*.q = 45 ∧ *staHead* = $y.q$.

Combining these results by Rules 5 and 2, we get

$\{\{q\}, Coll, Distr\}$: *true* o→ *pc*.q = 45 ∧ *staHead* = $y.q$.

Using the PSP-rule and (w0), we then obtain

$\{\{q\}, Coll, Distr\}$: *nx.staHead* = m
o→ (*pc*.q = 45 ∧ *staHead* = $y.q$ ∧ *nx*.($y.q$) = m) ∨ R .

One can verify by means of Rule 1 that, for $45 \leq s \leq 52$,

$\{\{q\}\}$: *pc*.q = s ∧ *staHead* = $y.q$ ∧ *nx*.($y.q$) = m
o→ (*pc*.q = s + 1 ∧ *staHead* = $y.q$ ∧ *nx*.($y.q$) = m) ∨ R .

By Rule 1, we also have

$\{\{q\}\}$: *pc*.q = 53 ∧ *staHead* = $y.q$
∧ *nx*.($y.q$) = m o→ R .

Finally, repeated application of Corollary 0 (and Rule 2) yields formula (10). This concludes the proof of (6), and thus the proof of progress formula (0).

7 Conclusion

We have developed the concept of bounded fairness in order to combine the assets of wait-freedom with the possibility to delegate tasks to separate processes. This enables a separation of concerns that can be crucial for succesful design.

We have developed and applied a variation of the logic of UNITY to prove that a system of sequential processes that communicate via shared memory satisfies progress assertions under bounded fairness. This extends the applicability of the UNITY approach, but it does not directly address the methodological challenge to prove progress properties in a systematic manner. The additional cost to prove the stronger property of *bounded* delay turns out to be small.

The application presented is sufficiently complicated to conclude that our method for expressing and proving progress is applicable to nontrivial systems.

Acknowledgments. Constructive criticisms of the referees have led to considerable improvements in the presentation.

References

1. A. Arora: Efficient reconfiguration of trees: a case study in methodological design of nonmasking fault-tolerant programs. In: H. Langmaak, W.-P. de Roever, J. Vytopil (eds.): Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer 1994 (LNCS 863). pp 110–127
2. K.M. Chandy, J. Misra: Parallel Program Design, A Foundation (Addison–Wesley, 1988)
3. N. Francez: Fairness. Springer, Berlin Heidelberg New York 1986
4. M.P. Herlihy: Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13**: 124–149 (1991)
5. M.P. Herlihy, J. Wing: Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst **12**: 463–492 (1990)
6. W.H. Hesselink: Wait-free linearization with an assertional proof. Distributed Computing **8**: 65–80 (1994)
7. W.H. Hesselink: Theories for mechanical proofs of imperative programs. Formal Aspects of Computing **9**: 448–468 (1997)
8. W.H. Hesselink: The design of a linearization of a concurrent data object. In: D. Gries, W.-P. de Roever (eds.): Programming Concepts and Methods, Proceedings Procomet '98, Chapman & Hall, IFIP 1998, pp 205–224
9. W.H. Hesselink: Progress for memory management of a concurrent data object. To be obtained from [10]
10. W.H. Hesselink: Web site: http://www.cs.rug.nl/~wim/linproc
11. S. Owicki, D. Gries: An axiomatic proof technique for parallel programs. Acta Informatica **6**: 319–340 (1976)
12. F.W. Vaandrager: Verification of a distributed summation algorithm. In I. Lee, S.A. Smolka (editors): Proceedings 6th International Conference on Concurrency Theory (Concur'95). Springer V., 1995 (LNCS 962), pp 190–203

Wim H. Hesselink received his Ph.D. in mathematics from the University of Utrecht in 1975. After ten years of research in algebraic groups and Lie algebras he turned to computing science. In 1986/1987 he was on sabbatical leave with the University of Texas at Austin. In 1994, he became a professor for the Groninger Universiteitsfonds in the field of program correctness. After that, he was for some years Chairman of the Department of Computing Science at the University of Groningen. His research interests include predicate transformer semantics, aspects and modalities of nondeterminacy, design and correctness of all kinds of algorithms, and the use of a mechanical theorem prover for the design of distributed systems.